

Elemental functions: Writing data-parallel code in C/C++ using Intel® Cilk™ Plus

A simple C/C++ language extension construct for data parallel operations

Robert Geva robert.geva@intel.com

Introduction

Intel® Cilk™ Plus provides simple to use language extensions to express data and task-parallelism to the C and C++ language implemented by the Intel® C++ Compiler, which is part of Intel® Parallel Composer and Intel® Parallel Studio. This article describes one of these programming constructs: “elemental functions”.

There are cases in which the algorithm performs operations on large collection of data, with no dependence among the operations being performed on the various data item. For example, the programmer may write, at a certain point of the algorithm: add arrays a1 and a2 and store the result in array r. When thinking at that level, the programmer is thinking in terms of a single operation that needs to be performed on many elements, independently of each other. Unfortunately, the C/C++ programming languages do not provide constructs that allow expressing the operation as such. Instead, they force the programmer to elaborate the operation in terms of a loop that operates on each array element. The end result, in terms of the values being stored in the result array, would be the same. However, the introduction of the loop introduces an unintended order of operations: The implementation has to add the first two array elements, store the results in the first location of the result array, move on to perform the same operation on the second set of elements, and so on. An example later will show what is expected to be a typical use of elemental function: use a single line to mark a standard C/C++ function as elemental, change the invocation loop from serial to parallel (in this example, it is a change of one keyword) and that is it. In the example below, there is a measured speed up of more than 15X¹.

Limitations of serial programming

The serial program under utilizes the parallel HW resources and therefore in most cases performs slower than an equivalent parallel program. Moreover, in many cases, the C/C++ language semantics do enforce the serial execution of the program and do not allow an optimizing compiler to implement a parallel version of the serial program. A compiler can only provide a parallel implementation of a

¹ Compiler: Intel® Parallel Composer 2011 with Microsoft* Visual Studio* 2008
Compiler options: /c /Oi /Qip /D "WIN32" /D "NDEBUG" /D "_CONSOLE" /D "_UNICODE" /D "UNICODE" /EHsc /MD /GS /fp:precise /Fo"Release/" /Fd"Release/vc90.pdb" /W3 /nologo /Zi
System Specifications: Intel® Xeon® E5462 processor, 2.8 GHz, 8 cores, 8 GB RAM, Microsoft Windows Server 2003* x64 Edition.
Sample code available at <http://software.intel.com/en-us/articles/intel-cilk-plus-black-scholes-sample/>

portion of a program if it can be proven in compile time that the parallel implementation and the serial implementation would be equivalent. The success of these proofs depends on many variables, for example the language construct being chosen to implement the code. In the sample code below, the compiler cannot prove that the arrays pointed- to by the 3 pointers which are provided as function argument – do not point into the same arrays.

```
void do_add_doubles(double * src1, double * src2, double * r, int length)
{
    for (i = 0; i < length; ++i) {
        *r = *src1 + *src2;
    }
}
```

Array notations

Intel Cilk Plus provide simple language constructs that operate on arrays, instead of requiring the programmer to express the algorithm in terms of loops that operate on elements of arrays. The example of adding two arrays can be expressed in Intel Cilk plus as

```
r[0:N] = src1[0:N] + src2[0:N];
```

In this example, the elements of the arrays src1 and src2 are added and the results are assigned to array r. The usual subscript syntax in C/C++ is replaced by an array section descriptor. In this example, 0:N means that the addition uses N elements of the array, starting from index number 0. The important difference between using this language construct and the standard C/C++ loop is that here, there is no serial ordering implied. Therefore, the expected code generation strategy from the compiler is vector parallelism, i.e. use the SSE instructions to implement the additions in a SIMD fashion. The compiler generates vector, SIMD code for operations on arrays including all the language built- in operators, such as '+', '*', '&', '&&' etc. However, the most general operation is when the programmer describes the operation using a function.

Elemental functions

An elemental function is a regular function, which can be invoked either on scalar arguments or on array elements in parallel.

So far, the Intel compilers have been providing a limited form of this functionality, some of the language standard library functions. For example, the programmer can write code in a loop that calls certain math library functions such as a sin() function, and the compiler can automatically vectorize the loop.

```

for (i = 0; i < length; ++i) {
    a[i] = b[i] * c[i];
    d[i] = sin(a[i]);
    e[i] = d[i] * d[i];
}

```

The implication is that the code can execute a short vector version of the sin function that takes a number of arguments which depends on the vector length of the HW XMM registers (e.g. 4) and compute 4 results in one invocation, at about the same time it would take to compute a single result.

The new “elemental functions” feature extends the capability from built in functions to application code functions. The programmer can now provide a function and use a `__declspec (vector)` annotation to indicate to the compiler to generate a short vector version of the implementation of the function. Note that the compiler also generates a scalar implementation of the function, and the program can invoke the function either on single arguments or arrays of argument, at different places in the program. Lastly, the program needs to invoke the elemental function in a parallel context, i.e. provide multiple arguments instead of a single one. The parallel invocation can be done in a number of ways.

1. Invoke the function from a C/C++ standard for loop. With this invocation, the compiler will replace the function call with a call to the short vector version within the loop.

```

for (i = 0; i < length; ++i) {
    res[i] = elemental_func(a[i],b[i]);
}

```

With this invocation, the instances of the elemental function will all execute on a single thread.

Note that as a programmer, you do not need to take care of the case where the number of loop iterations is not evenly divided by the number of arguments handled by the short vector version. The compiler takes care of the remainder automatically.

2. Invoke the function from a `cilk_for` loop:

```

_cilk_for (i = 0; i < length; ++i) {
    res[i] = elemental_func(a[i],b[i]);
}

```

With this invocation, multiple instances of the elemental functions will be spawned into execution cores and execute in parallel. Therefore, with a single expression of the parallel work, you get the benefit of both multi core parallelism and vector parallelism.

3. Invoke the function from array notation syntax.

```

res[:] = elemental_func(a[:] , b[:]);

```

With this invocation syntax, the the current compiler implementation provides the same behavior as the serial loop in case 1 above. However, this syntax is reserved for future implementations to provide an optimal mix of both behaviors.

The Task Scheduler

When instances of an elemental function are spawned, the compiler is using the run time task scheduler, which is the same task scheduler that manages other work spawned with the `_Cilk_spawn` and `_Cilk_for` constructs. The scheduler is a user level (not OS level) work stealing task scheduler that does parent stealing. Work stealing means that when a code stream executes a task spawn, it queues work in its own work deque. Workers do not push work onto other workers. A worker only steals work from another worker if its own work deque is empty. This policy ensures load balancing and prevents over subscription of workers. “Parent stealing” means that when a code sequence executes the statement “`_Cilk_spawn func(y);`” then the same worker continues to execute the function `func`, and the continuation of the spawning function is what is detached, placed on the deque and made available for stealing. The worker executes “`func`”, and if no other worker stole it, then it returns from the execution of “`func`” and pops the continuation from its own work deque. The end result is that the behavior matches that of a regular function call in serial execution. The task scheduler’s policy guarantees that the program produces the same result if executing on a single or multiple cores. The combination of very low overhead during spawning a task with the automatic load balancing suggests a more natural approach to the overall SW architecture of a parallel program. Some parallel programs are architected with a monolithic perspective in which an architect carefully orchestrates the work being done at each point of the program by each thread. The program only allows as much parallelism as the target platforms supports. By contrast, with the work stealing task scheduling approach, parallelism does not have to be orchestrated globally throughout the program. It can be originated independently from multiple modules of the program with low risk of overwhelming execution resources. Conversely, making more parallelism available than what the current platform can benefit from, may lead to a scalable program that can be moved over time to platforms with more execution resources and benefit from the additional parallelism.

Performance With Intel® Cilk™ Plus Elemental Functions

With this introduction of elemental function, let’s take a look at the Black-Scholes example² using elemental functions. Black-Scholes is a well known short program used for pricing of European style options. In this example, the main loop is in the function `test_option_price_call_and_put_black_scholes_Serial()` which, in turn, invokes the call and put options calculations using the original or serial version of the routines:

² Sample code available at <http://software.intel.com/en-us/articles/intel-cilk-plus-black-scholes-sample/>

This sample is derived from code published by Bernt Arne Odegaard, http://finance.bi.no/~bernt/gcc_prog/recipes/recipes/

```

double option_price_call_black_scholes(...) {...}
double option_price_put_black_scholes(...) {...}

void test_option_price_call_and_put_black_scholes_Serial(
    double S[NUM_OPTIONS], double K[NUM_OPTIONS],
    double r, double sigma, double time[NUM_OPTIONS],
    double call_Serial[NUM_OPTIONS], double put_Serial[NUM_OPTIONS])
{
    for (int i=0; i < NUM_OPTIONS; i++) {
        // invoke calculations for call-options
        call_Serial[i] = option_price_call_black_scholes(S[i], K[i], r, sigma, time[i]);

        // invoke calculations for put-options
        put_Serial[i] = option_price_put_black_scholes(S[i], K[i], r, sigma, time[i]);
    }
}

```

are called. An array- notation call syntax is below:

```

__declspec(vector) double option_price_call_black_scholes(...) {...}
__declspec(vector) double option_price_put_black_scholes(...) {...}

void test_option_price_call_and_put_black_scholes_ArrayNotations(
    double S[NUM_OPTIONS], double K[NUM_OPTIONS],
    double r, double sigma, double time[NUM_OPTIONS],
    double call_ArrayNotations[NUM_OPTIONS], double put_ArrayNotations[NUM_OPTIONS])
{
    // invoking vectoried version of the call functions optimized for given core's SIMD
    call_ArrayNotations[:] = option_price_call_black_scholes_ArrayNotations(S[:], K[:], r, sigma, time[:]);

    // invoking vectoried version of the put functions optimized for given core's SIMD
    put_ArrayNotations[:] = option_price_put_black_scholes_ArrayNotations(S[:], K[:], r, sigma, time[:]);
}

```

As noted before, the current Intel compiler does not yet implement the capability to spawn multiple calls to the elemental function into cores. That can be accomplished using the `_Cilk_for` loop calling syntax, as shown below:

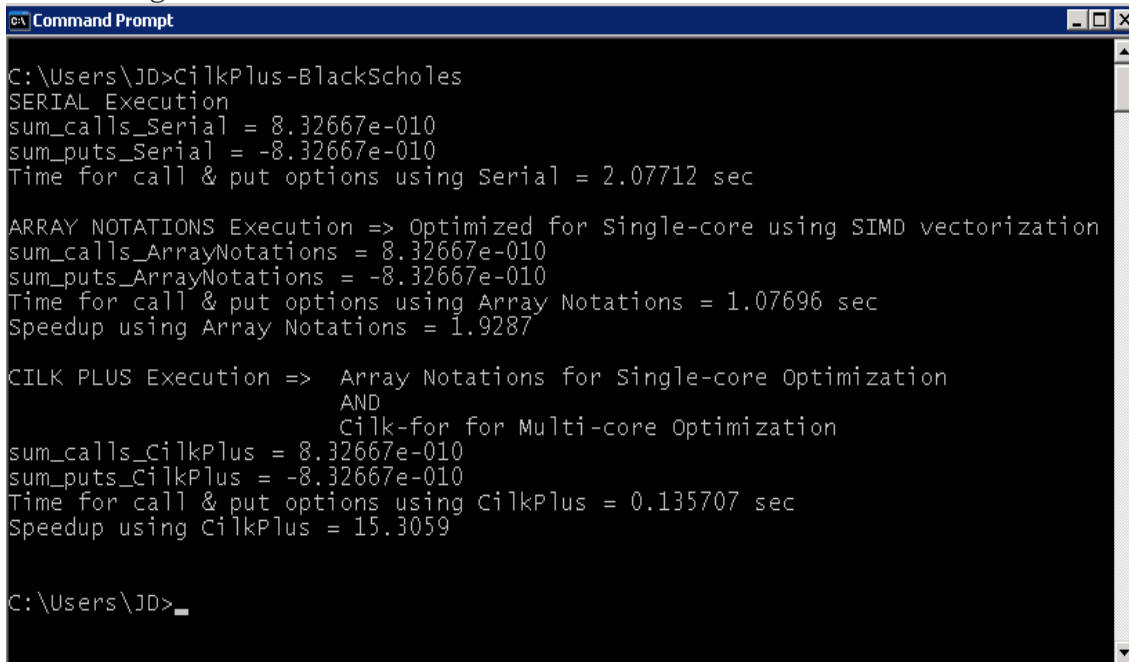
```

void test_option_price_call_and_put_black_scholes_CilkPlus(
    double S[NUM_OPTIONS], double K[NUM_OPTIONS],
    double r, double sigma, double time[NUM_OPTIONS],
    double call_CilkPlus[NUM_OPTIONS], double put_CilkPlus[NUM_OPTIONS])
{
    // invoking vectoried version of the call and put functions optimized for given core's SIMD
    // and then spreading it across mutple cores using cilk_for for optimized multi-core performance
    cilk_for (int i=0; i<NUM_OPTIONS; i++) {
        call_CilkPlus[i] = option_price_call_black_scholes_ArrayNotations(S[i], K[i], r, sigma, time[i]);
        put_CilkPlus[i] = option_price_put_black_scholes_ArrayNotations(S[i], K[i], r, sigma, time[i]);
    }
}

```

This simple, yet powerful, change automatically parallelizes the loop and executes the function calls on multiple cores. Note that the Cilk Plus runtime determines the number of cores available on the system and distributes the load accordingly.

Running this optimized parallel binary on a 8-core machine gets us a combined speedup of $15.31x^3$ over the original serial version:



```
Command Prompt
C:\Users\JD>CilkPlus-BlackScholes
SERIAL Execution
sum_calls_Serial = 8.32667e-010
sum_puts_Serial = -8.32667e-010
Time for call & put options using Serial = 2.07712 sec

ARRAY NOTATIONS Execution => Optimized for Single-core using SIMD vectorization
sum_calls_ArrayNotations = 8.32667e-010
sum_puts_ArrayNotations = -8.32667e-010
Time for call & put options using Array Notations = 1.07696 sec
Speedup using Array Notations = 1.9287

CILK PLUS Execution => Array Notations for Single-core Optimization
                        AND
                        Cilk-for for Multi-core Optimization
sum_calls_CilkPlus = 8.32667e-010
sum_puts_CilkPlus = -8.32667e-010
Time for call & put options using CilkPlus = 0.135707 sec
Speedup using CilkPlus = 15.3059

C:\Users\JD>
```

³ See footnote 1 for configuration information.

Optimizations

In some cases, when the function is applied to successive elements of an array, one or more of its arguments takes the same value across all invocations. We call this a scalar argument. An example is that if instead of adding two arrays and storing the result in a third array, you want to add one value to all elements of an array, and store the results in a result array. You can write an elemental function to do that using the syntax shown above.

```
__declspec (vector) double ef_add_doubles(double x, double a)
{
    return x + a;
}
//invoke the function
for (i = 0; i < n; ++i) {
    y[i] = ef_add_doubles(x[i],42);
}
```

While this will work and produce the expected results, it is a somewhat wasteful. If the function is designed to add the same value to all elements of an array, then there is no need to pass the value multiple times. The above implementation will pass the value 42 along to `ef_add_doubles` together with each value of an element of the array `x`, whereas passing it only once would be sufficient. The way to indicate to the compiler that passing the value once instead of multiple times is sufficient is by using a the scalar clause in the `declspec` vector:

```
//each invocation passes a short vector of elements of x and a single value of a
__declspec (vector scalar(a)) double ef_add_doubles(double x, double a)
{
    return x + a;
}
//invoke the function
for (i = 0; i < n; ++i) {
    y[i] = ef_add_doubles(x[i],42);
}
```

Another optimization opportunity is when the value of an argument is not the same for each invocation, but instead changes by a constant increment each time. For example, you can add two arrays and store the result into a result array by passing the start addresses of the arrays to the elemental function, along with an index variable. In this case, the value of the variable `i` is incremented consistently each time, and therefore does not need to be passed multiple times to the function.

```
__declspec (vector (linear (r,a,b))) void ef_add_doubles(int i)
{
    r[i] = a[i] + b[i];
}
//invoke the function
for (i = 0; i < n; ++i) {
    ef_add_doubles(&r[0],&a[0],&b[0],i);
}
```

Related constructs

Functions calls can be executed in parallel without being elemental. With the current Intel compiler, you can invoke a function call from a `_Cilk_for` loop, and the iterations will execute in parallel. You can also use the array notations to map a scalar function onto elements of an array, for example:

```
res[:] = some_func(a[:] , b[:]);
```


This construct applies the function to elements of the input array and assigns the results to the corresponding elements of the output array. This construct is defined such that there is no enforced ordering between the execution of the function on the array elements, and therefore multiple instances can be invoked in parallel.

The advantage of using the elemental function construct over the above is that you benefit from both types of parallel HW resources, the cores and the vectors.

Language Rules

The compiler imposes several limitations on the language constructs that are allowed to be used inside elemental functions. The purpose of these rules is to guarantee that the compiler can generate a short vector implementation of the function. Some of these rules are artifacts of the current state of the technology and will be removed in future implementation, as the capability improved. Currently, the following are disallowed inside elemental functions:

- Loops, including the keywords for, while, do and overloaded operators
- Switch statements
- Goto, setjmp, longjmp
- Calls to functions other than other elemental functions and the intrinsic short vector math libraries provided with the Intel compilers
- Operations on structs, other than member selection
- `_Cilk_spawn`
- Array notations
- C++ exceptions
- Elemental functions cannot be virtual, and can only be called directly, not through a function pointer

Note, however, that elemental functions do not have to be pure functions. They are allowed to modify global state and have side effects. They can make recursive function calls.

Summary

When your algorithm requires performing the same operation on multiple data items, and the data items can be organized in an array, consider using the new language construct provided with Intel® Cilk Plus™ - elemental functions. Writing an elemental function is simply writing a scalar function, followed by invoking it in a parallel context as shown above. The Intel compiler will provide an implementation that gives you the benefits of maximizing both the SIMD vector capabilities of your processor and the multiple cores with little effort to develop your parallel code and maintain it.

Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the "Intel® Compiler User and Reference Guides" under "Compiler Options." Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101